SYSTEM AND METHOD FOR USING NATIVE CODE INTERPRETATION TO MOVE THREADS TO A SAFE STATE IN A RUN-TIME ENVIRONMENT

Inventors: Joakim Dahlstedt
Peter Lonnebring

Claim of Priority:

[0001] This application claims the benefit of U.S. Provisional Application "SYSTEM AND METHOD FOR USING NATIVE CODE INTERPRETATION TO MOVE THREADS TO A SAFE STATE IN A RUN-TIME ENVIRONMENT", Serial No. 60/434,785; filed December 18, 2002, and incorporated herein by reference.

Field of the Invention:

[0002] The invention is generally related to run-time and virtual machine environments, and particularly to a system and method for moving threads to a safe state within the run-time environment using native code interpretation.

Background of the Invention:

[0003] As described herein, a run-time environment is typically one in which an application server or virtual machine executes on a computer processing device, computer, or machine. Examples of such run-time

environments include Sun's Java Virtual Machine from Sun Microsystems, Inc. and the JRockit Virtual Machine from Bea Systems, Inc.. Typically the run-time environment, often referred to as a virtual machine (VM) executes upon an application server operating system, and provides access to a plurality of clients such as Java clients, that wish to run applications at the server. The system is generally thread-based in that requests for processing at the server execute in threads and are handled by the server within these threads. However, problems may arise if the threads are stopped in an unstable (or more correctly termed "unsafe") state. The term "run-time system" may suggest that threads are continuously running or executing. However, there are many instances, some of which are described in further detail below, in which the threads must be stopped in order to perform some particular operation. When the threads are stopped in this manner problems may arise unless the threads are stopped in a safe state.

[0004] Among the circumstances that require the threads to be stopped in order to perform operations, the following are perhaps of most importance.

1. Garbage collection. Most run-time environments, including the Java Virtual Machine and the JRockit Virtual Machine support some form of garbage collection. This is a process in which objects that are no longer in use by the run-time environment are deleted or cleared, to free up resources for use by other objects. A typical garbage collection process requires each thread to report which objects associated with that thread are still in use or "alive". The usual approach to this is to create a "live map" or a garbage collection map for each thread which identifies the live objects. However, in order to create the live map or garbage collection map the thread must be stopped in a safe state, and preferably on an instruction that contains a live object.

2. Context switching. Depending on the run-time environment or the virtual machine being used, context switching may be performed within the threads. Preemptive switching from one thread to another is not always allowed, and depending on the run-time environment may not be recommended. In any case context switching is only allowed on a safe state instruction. This requires the threads to be stopped in a safe state prior to context switching. Some vendors have tried to work around this problem by incorporating a state flag within each thread. For example, operating systems such as Solaris and Linux use a state flag to indicate whether the thread is in a non-switchable state. If the state flag indicates the thread is in a non-switchable state the system resumes thread execution and retries again at a later point in time.

3. Thread locking on a single CPU machine. This factor is an extension of the context switching feature described above. When using a single processor machine (i.e., using a single thread to run all of the code), care must be taken to ensure that preemptive thread switching is not performed within lock regions.

[0005] The traditional approach to the problem of stopping the threads in a safe states is to select one of three strategies. First, the system can attempt to make all states safe prior to stopping them. This is not always possible depending on the specific situation. This approach is also very costly memory-wise to implement.

[0006] An alternative approach is to poll the threads, i.e., to stop, then restart the thread, and stop it again a little later in time, until it is eventually stopped in a safe state. The problem with this approach is that it is cumbersome and involves numerous thread context switching. Furthermore, the polling approach is not even guaranteed to terminate since it starts and stops threads randomly and will only complete if it stops in a safe state, which potentially may never happen (although will typically just take a long time to happen).

[0007] The third approach is to determine if a thread is stopped in an unsafe state, and if it is determined as such then the system inserts a stop-point at a safe-state point, restarts the thread, and lets it run until it reaches the stop-point. This code patching approach is also very cumbersome and time-consuming to use, and requires numerous thread context switches.

Summary of the Invention:

[0008] As described herein the present invention provides a system and a method for interpreting native code to move threads to a safe state in a run-time environment and which overcomes many of the problems associated with traditional approaches, including thread polling, and code patching. In a run-time or virtual machine environment, threads are used to process requests to the virtual machine (VM). In many instances such as garbage collection, context switching, and single CPU locking, the threads must be stopped in a safe state in order for the operation to successfully complete. The invention can be used to ensure that a thread is stopped in such a safe state. In accordance with an embodiment of the invention, when a first thread A is stopped by a second thread B, if A is not in a safe state the invention allows thread B to roll thread A forward to a safe state by interpreting the machine instruction currently at A. A's state is

then updated accordingly. When the system is satisfied that A is standing at a safe state instruction, the machine instruction interpretation can stop.

Brief Description of the Figures:

[0009] **Figure 1** shows a flowchart of a process in accordance with an embodiment of the invention that allows a first thread to halt execution of a second thread such that the second thread is stopped in a safe state.

[0010] **Figure 2** shows a schematic of a system in accordance with an embodiment of the invention that uses a native code interpreter within a VM to perform thread stopping such that the threads are stopped in a safe state.

[0011] **Figure 3** illustrates a schematic of an embodiment of the invention that illustrates how a first thread A can be stopped and then rolled forward by a second thread B such that thread A is stopped in a safe state.

[0012] **Figure 4** illustrates a flowchart in accordance with an embodiment of the invention that illustrates how a thread uses machine instruction interpretation to roll another thread forward to a safe state.

Detailed Description:

[0013] The present invention provides a system and a method for interpreting native code to move threads to a safe state in a run-time environment. In a run-time or virtual machine environment, threads are used to process requests to the virtual machine (VM). In many instances such as garbage collection, context switching, and single CPU locking, the threads must be stopped in a safe state in order for the operation to successfully complete. The invention can be used to ensure that a thread is stopped in such a safe state. In accordance with an embodiment of the invention, when a first thread A is

stopped by a second thread B, if A is not in a safe state the invention allows thread B to roll thread A forward to a safe state by interpreting the machine instruction currently at A. A's state is then updated accordingly. When the system is satisfied that A is standing at a safe state instruction, the machine instruction interpretation can stop. The system can be used with multiple threads, and can be implemented to solve many types of problems in addition to context switching and garbage collection.

[0014] **Figure 1** shows a flowchart of a method in accordance with an embodiment of the invention in which a first thread is allowed to stop a second thread such that the second thread is in a safe state. Stopping threads in this manner allows certain operations such as garbage collection, context switching and thread locking to take place, which would otherwise not be possible. As shown in **Figure 1**, in step **2** the system allows the threads to execute normally. This is the typical situation in a virtual machine environment. In step **4**, when the threads need to be stopped, for example, for purposes of garbage collection, a first thread (the stopping thread) is allowed to halt the execution of a second thread (the executing thread) within the virtual machine. In step **6**, the system then determines if the executing thread is in a safe state. If the system determines that the executing thread is currently in a safe state then it is stopped, as indicated by step **8**. However, if the system determines in step **10** that the executing thread is not currently in a safe state, then the stopping thread is allowed to roll forward the executing thread until it is finally determined to be in a safe state. In practice, this roll forward is performed in a series of iterations, including at each point in step **6** determining if the executing thread is currently in a safe state. When, in step **6**, the system eventually decides that the executing thread is now in a safe state, the process ends.

[0015] **Figure 2** illustrates a schematic of a typical system in accordance with an embodiment of the invention that may use or incorporate the current invention. As shown in **Figure 2**, the system **18** includes a server **20** which may be a physical computer, computing device or a machine, or may be a server process running on a computer. The server will typically include hardware **22** and operating system **24** components, in addition to a virtual machine or run-time environment **26**. As used in this embodiment of the invention the run-time environment includes application code **28** which executes as a number of threads **30**, including a first thread A **32** and a second thread B **34**. As used in this embodiment of the invention, the system also includes a native code interpreter **36** which the system uses to allow a thread to roll forward another thread to a safe state by interpreting the native code and the state at which the currently executing thread exists.

[0016] **Figure 3** illustrates a schematic in accordance with an embodiment of the invention showing how a thread may stop execution, and subsequently roll forward the executing thread so that executing thread is stopped in a safe state. As shown in **Figure 3**, the typical situation with the run-time environment is to have threads A **32** and threads B **34** currently executing (indicated by the arrows **40**and **42** respectively). At any given point in time, for example when garbage collection is initiated or when the system wishes to perform context switching within the threads, one of the threads, in this instance thread B, is stopped (indicated by circle **44**). In accordance with an embodiment of the invention, thread B could also request that thread A be stopped. When thread A is first stopped, the system determines if thread A is currently in a safe state. If its not in a safe state then thread B is allowed to roll forward thread A using a machine code interpretation **46** of thread A's current state. Thread A is

subsequently rolled forward in discrete steps (indicated by arrows **48**, **50**, and **52** respectively), until thread A is found to be in a safe state. When both threads A and B are stopped at a safe state (indicated by circles **44** and **54**) the process ends.

[0017] **Figure 4** illustrates a flowchart in accordance with an embodiment of the invention that allows an executing thread to be stopped by another thread and then rolled forward using machine instruction interpretation so that the thread is stopped in a safe state.

[0018] In step **10**, which corresponds to step **10** in **Figure 1**), the system determines that if the executing thread is not currently in a safe state it should allow a stopping thread to roll the executing thread forward to a safe state. In step **62** the system interprets the machine instruction currently at A. In step **64** A's state is then updated in response to interpreting the machine code instruction. In step **66**, control is then returned to the main process to determine whether the executing thread is now in a safe state and thus can be safely stopped.

[0019] The present invention may be conveniently implemented using a conventional general purpose or a specialized digital computer or microprocessor programmed according to the teachings of the present disclosure. Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art.

[0020] In some embodiments, the present invention includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the processes of the present invention. The storage medium can include, but is not

limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, microdrive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0021] The foregoing description of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. Particularly, while the embodiments of the system and methods described above are described in the context of a WebLogic server, and a JRockit virtual machine, it will be evident that the system and methods may be used with other types of application servers, runtime systems, and virtual machines, including other types of JVMs. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalence.